WhizzML Tutorials

The BigML Team

Version 1.18

MACHINE LEARNING MADE BEAUTIFULLY SIMPLE

info@bigml.com

Last updated March 28, 2024

About this Document

This document contains a collection of WhizzML tutorials, organized by complexity.
These are intended to help the user write his/her own WhizzML Scripts and Libraries.

Contents

Introduction

As you have read in the WhizzML Primer document,[1] WhizzML is a powerful tool for automating Machine Learning (ML) workflows and implementing higher-level ML algorithms.

This document contains tutorials for writing these WhizzML workflows and algorithms. The tutorials are organized into three categories: Part I, Part II and Part III.

In the beginner tutorials, we go over each piece of the code in detail, explaining WhizzML language concepts and its standard library. As the tutorials get more advanced, we leave these details behind to focus on higher level concepts.

[1] https://bigml.com/whizzml

Part I

Beginner

We'll start with a few basic tutorials that demonstrate just a few of the most basic features of WhizzML. It might be handy to have the WhizzML Reference nearby as you go through these examples.

CHAPTER 1

Model or Ensemble?

In this tutorial, we'll do a simple test to determine whether a single tree or an ensemble is a better fit for a given dataset. We'll go over every line of the code here, but you can also grab it from our WhizzML examples repository[1] on Github.

The idea here is simple: We can use WhizzML to build both single trees and ensembles of trees as predictors. BigML can also conduct evaluations of predictors on datasets. So our script will:

- Split our datasets into two parts, training and testing
- Train a single tree on the training data
- Train an ensemble of trees on the training data
- Evaluate both on the testing data
- Compare the evaluations to decide which is better

1.1 Creating the Training and Testing Sets

The first thing we need to do is split our datasets into two parts. Taken together, these two functions do exactly that.

```
;; Functions for creating the two dataset parts
;; and the model and ensemble from the training set.

(define (sample-dataset ds-id rate oob)
  (create-dataset ds-id {"sample_rate" rate
                         "out_of_bag" oob
                         "seed" "whizzml-example"}))

(define (split-dataset ds-id rate)
  (list (sample-dataset ds-id rate false)
        (sample-dataset ds-id rate true)))
```

As you can see, the `sample-dataset` function takes three parameters, `ds-id`, which is the identifier for the input dataset, `rate` which is the amount of the data that we want to keep, and `oob`, a Boolean value that says whether to use the sample specified, or its complement. That is, if `rate` is 0.75 and `oob` is false, we'll get 75% of the data in our sample. If `oob` is true, we'll get the other 25%.

This is important to our split as we want our training and testing data to be mutually exclusive (so we can't "cheat" by training on testing points before the test). In fact, you can see this at work in `split-dataset`. Here we return a list of two datasets created by `sample-dataset`, where the only

[1] https://github.com/whizzml/examples/tree/master/model-or-ensemble

parameter we've varied is `oob`. This will give us a two mutually exclusive datasets, one of which we'll use to train, and the other to test.

1.2 Creating Predictors

The next thing we need to do is create our single tree and our ensemble. Given a dataset identifier, `ds-id`, creating a model from it with default configuration parameters is as simple as calling the built-in procedure `create-model`:

```
(create-model ds-id)
```

For the ensemble, we have the corresponding `create-ensemble` primitive, although in this case we will want to also provide a configuration parameter: the number of models in the forest, `size`:

```
(create-ensemble ds-id {"number_of_models" size})
```

In both cases, the `create` procedure will first check that the dataset that is used has reached its `finished` state (i.e., has status code 5), waiting if necessary for it, and then request the creation of the corresponding model or ensemble. The result of the call will be the identifier of the new resource.

1.3 Evaluating Predictors

The last major step is to evaluate the models on the testing data. To that end, we can use the WhizzML built-in 'create-evaluation', that takes as inputs the identifier of the resource to evaluate (a model or an ensemble in our case) and the identifier of the test dataset to use for the evaluation. Thus, our code performing evaluations will be very straightforward:

```
(create-evaluation model-id test-dataset-id)
(create-evaluation ensemble-id test-dataset-id)
```

These calls will return the corresponding evaluation identifiers. We will want to, first, wait until they are finished, and, once they are completed, fetch them and extract from the full resource map the quantity that we are going to use to assess the quality of our model or ensemble, namely, the average F-measure. These two steps are encapsulated in the helper function `quality-measure`, that we define as follows:

```
(define (quality-measure ev-id)
  (let (ev (fetch (wait ev-id)))
   (ev ["result" "model" "average_f_measure"])))
```

As you can see, we first call `wait` with the evaluation's identifier to make sure the evaluation is finished before fetching it. `wait` will eventually return that identifier again if everything is fine, and then we use that identifier again as the argument of `fetch`, that recovers the full map of the evaluation. All that is left is to access the nested field "average_f_measure" within that map:

```
{"result" {"model" {"average_f_measure" 0.5
                    "average_phi" 0.6
                    ...}
           ...}
  ...}
```

As you can see, we use the evaluation map itself, `ev`, as a function to perform the lookup, passing to it the path to the desired key as a list, `["result" "model" "average_f_measure"]`.

1.4 The Whole Workflow

Finally, we're ready to put it all together.

```
;; Function encapsulating the full workflow
(define (model-or-ensemble src-id)
  (let (ds-id (create-dataset src-id)   ;; full dataset
        ids (split-dataset ds-id 0.8)   ;; split it 80/20
        train-id (ids 0)                ;; the 80% for training
        test-id (ids 1)                 ;; and 20% for evaluations
        m-id (create-model train-id)
        e-id (create-ensemble train-id {"number_of_models" 15})
        m-f (quality-measure (create-evaluation m-id test-id))
        e-f (quality-measure (create-evaluation e-id test-id)))
    (log-info "model f " m-f " / ensemble f " e-f)
    (if (> m-f e-f)  m-id e-id)))
```

The final function looks basically like our itemized list at the top of this tutorial. We first create a dataset using the input source identifier. We then split this dataset using the function `split-dataset` above that gives us a list of two sub-datasets. We can use the returned list as a procedure to pull out the first and second elements of this list into `train-id` and `test-id`, respectively. We then make our single tree and our ensemble and evaluate them both, pulling the value for the F-measure out of the evaluation.

Depending on which value for the F-measure is better, we return the identifier of the better thing, and we're done!

1.5 Extension to Regression

In the above code, we use a quality metric, the average value of F1, that is only available when our problem is a classification. For regressions (that is, models where our objective field is numeric), that metric is not available and our script will raise an error.

We can easily fix the problem and extend our script so that it supports regression problems by picking a convenient quality metric for them. A good choice is `r_squared`, which can be found nested following the path `["result" "model" "r_squared"]` in the evaluation map.

Instead of `f-measure` we will use a new function, `quality-measure` that will first try to find the average F measure and, in case it is not found, will fallback to `r_squared`:

```
;; Function to extract a quality measure from the evaluation results
(define (quality-measure ev-id)
  (let (ev (fetch (wait ev-id)))
    (or (ev ["result" "model" "average_f_measure"] false)
        (ev ["result" "model" "r_squared"] false)
        (raise "The models couldn't be evaluated"))))
```

The implementation of this function also shows a simple example of error signaling using `raise`.

1.6 Summing Up

We have seen an example of how we can use WhizzML to do simple model selection with some basic code and no external libraries. We have written our first simple functions to abstract away implementation details, such as `quality-measure`, seen how to create and fetch BigML resources and access its properties, and simple error handling.

CHAPTER 2

Dataset Transform

We'll start with a script that will remove a field from a dataset if it has "too much" missing data. As those of you who have dealt with production data know, sometimes there are fields which are missing a lot of data. So much so, that we want to ignore the field altogether.

BigML will automatically detect bad fields like this and ignore them automatically if we create a predictive model. But what if **we** want to specify the required "completeness" of the data field? How can we eliminate fields that have, say, 95% non-missing data?

We can use WhizzML!

Let's Do it! Use the WhizzML reference guide[1] if you need it along the way.

To reiterate our goal, we want to write a function that:

1. Given a dataset and a specified threshold (e.g. 0.95)
2. Returns a new dataset with only the fields that are more than 95% populated.

We can define the base function here.

2.1 Filtered-Dataset

```
(define (filtered-dataset dataset-id threshold)
  (create-and-wait-dataset {"origin_dataset" dataset-id
                            "excluded_fields" (excluded-fields
                                               dataset-id
                                               threshold)}))
```

Hey, slow down! Ok. Let's take it step-by-step.

We define a new function called `filtered-dataset` that takes two arguments: Our starting dataset `dataset-id`, and a `threshold` (e.g., 0.95)

```
(define (filtered-dataset dataset-id threshold) ...)
```

What do we want this function to do? We want it to return a new dataset, hence:

```
(create-and-wait-dataset ...)
```

But we don't just want any old dataset, we want one based off our old dataset:

[1] https://bigml.com/whizzml

```
"origin_dataset" dataset-id
```

And we also want to exclude some fields from our old dataset!

```
"excluded_fields" (...)
```

Ah, but which fields do we want to exclude? We can let a new function called `excluded-fields` figure that out for us.

But for now, all we need to know is that this new function (`excluded-fields`) takes two arguments: our old dataset and our specified threshold.

The line above becomes: (indentation removed for clarity)

```
"excluded_fields" (excluded-fields dataset-id threshold)
```

As we progress, keep in mind that we want this new function (`excluded-fields`) to return a list of field names (e.g. `["field_1" "field_2" "field_3"]`)

Great! We defined our base function. Now we have to tell our new function, `excluded-fields` how to give us the list that we want.

2.2 Excluded-Fields

```
(define (excluded-fields dataset-id threshold)
  (let (data (fetch dataset-id)
        all-field-names (data "input_fields")
        total-rows (data "rows"))
    (filter (lambda (field-name)
              (> threshold (present-percent data field-name total-rows)))
            all-field-names)))
```

Wow what? You can use that code for reference, but don't be intimidated. We'll go over each piece.

First we define the function, declaring its two arguments: our original dataset, and the threshold we want to use.

```
(define (excluded-fields dataset-id threshold) ... )
```

Before we write any more code, let's talk about the meat of this function. We want to look at all the columns (fields) of this dataset, and find the ones that are missing too much data. We'll keep the names of these "bad" fields so that we can exclude them from our new dataset.

To do this, we can use the function `filter`. It takes two arguments: a list and a predicate. `filter` will return a new list composoed of the elements in the original list that satisfy the predicate. In our case, the predicate is that *the field has to have at least 95% of the data.*

```
(filter predicate    all-field-names)
;;      ^ function   ^ list
```

The first argument given to `filter` is our predicate. The predicate should be a function that either evaluates to `true` or `false` based on each element of the list we pass to it. If the predicate returns `true`, then that element of the list is kept. Otherwise, it is thrown out.

We can define this predicate function using `lambda`.

`lambda` is like any other function definition. We have to tell it the name of the thing we are passing into it

```
(field-name)
```

and also tell it what we are going to do with that thing.

```
(> threshold <percent-of-data-that-the-field-has>)
```

In our case, we are checking to see if the threshold is *greater than* the amount of data present. We will keep the `field-name`(s) that *do not* have enough data. (Because remember, these are the fields that will be excluded from our new dataset!)

Cool! But two things are still missing from our `filter`.

1. `all-field-names`
2. `<percent of data that the field has>`

How do we get these?

The first isn't too difficult because BigML datasets have this information readily available. We just have to `fetch` it from BigML first.

```
(fetch dataset-id)
```

and then specify which value we want to `get`.

```
((fetch dataset-id) "input_fields")
```

Nice.

To figure out what percent of the rows are populated for a specific field, we get to... Define a new function!

But before we do that, let's talk about some things we skipped over in our `excluded-fields` function. Here it is again, for convenience.

```
(define (excluded-fields dataset-id threshold)
(let (data (fetch dataset-id)
      all-field-names (data "input_fields")
      total-rows (data "rows"))
  (filter (lambda (field-name)
            (> threshold (present-percent data field-name total-rows)))
          all-field-names)))
```

What is `let`?

`let` is the preferred method for declaring local variables in WhizzML.

- We set the value of `data` to the result of `(fetch dataset-id)`.
- We set the value of `all-field-names` to the result of `(get data "input_fields")`.
- We set the value of `total-rows` to the result of `(get data "rows")`. (We didn't talk about this yet. It's one of the values we need to pass to the `present-percent` function)

`let` is useful for a couple of reasons in this function. First, we use `data` twice. So we can avoid the repetition of writing `(fetch dataset-id)` twice. Second, naming these variables at the top of the function makes the rest much easier to read!

So to wrap up this `excluded-fields` function, let's talk through what it does again. First, it declares local variables that we'll need. Then, it takes the list of `all-field-names` and filters it based on a function that checks its "present percent" of data points. We keep the names of the fields that *do not* meet our criteria. Cool!

Now, we'll go over that `present-percent` function

2.3 Present-Percent

```
(define (present-percent data field-name total-rows)
  (let (fields (data "fields"))
    (- 1 (/ (missing-count field-name fields) total-rows))))
```

Ah. Not so bad.

To calculate the percentage of data points that are present in a given field, we need a few things. First of all, the big collection of data from our dataset (`data`). Second, the name of the field we are inspecting (`field-name`). Last, the total number of rows in our dataset (`total-rows`).

We'll set another local variable using `let` and call it `fields`. This is another object containing data about each of the fields. We'll be using it below.

```
(let (fields (data "fields")) ...)
```

Then, we divide the `missing-count` from the field by the `total-rows`. This gives us a "missing percent"

```
(/ (missing-count field-name fields) total-rows))
```

We subtract the "missing percent" from one and that gives us the "present percent"!

```
(- 1 (/ (missing-count field-name fields) total-rows))
```

But `missing-count` is another function! Yes it is!

2.4 Missing-Count

```
(define (missing-count field-name fields)
  (fields [field-name "summary" "missing_count"]))
```

`missing-count` takes two arguments. First, the name of the field we are inspecting (`field-name`) and second, the `fields` object we mentioned earlier. It holds a bunch of information about each of the dataset fields.

To get the count of missing rows of data from the field, we do this:

```
(fields [field-name "summary" "missing_count"])
```

It lets us access an inner value (i.e. 10) from a data object structured like so:

```
fields = {field-name:
              {"summary":
                    {"missing_count": 10
                     "tags": ["cool" "fun"]}}
          something-else: ...}
```

And... That's it! We have now written all the pieces to make our `filtered-dataset` function work!

All together, the code should look like this:

```
(define (missing-count field-name fields)
  (fields [field_name "summary" "missing_count"]))
```

```
(define (present-percent data field-name total-rows)
  (let (fields (data "fields"))
    (- 1 (/ (missing-count field-name fields) total-rows))))

(define (excluded-fields dataset-id threshold)
  (let (data (fetch dataset-id)
        all-field-names (data "input_fields")
        total-rows (data "rows"))
    (filter (lambda (field-name)
              (> threshold (present-percent data field-name total-rows)))
            all-field-names)))

(define (filtered-dataset dataset-id threshold)
  (create-and-wait-dataset {"origin_dataset" dataset-id
                            "excluded_fields" (excluded-fields
                                               dataset-id
                                               threshold)}))
```

And we can run it just like this:

```
(filtered-dataset "dataset/83hs0sj3819sdkd92k" 0.75)
```

And get a result like this:

`"dataset/..."` - a new dataset without those empty fields.

Part II

Intermediate

Now that we've mastered the basics, let's take a look at some more involved workflows. Here, we'll be dealing with multiple resource types and using many of the standard library functions to accomplish our task.

CHAPTER 3

Covariate Shift

If this is your first time writing WhizzML, we suggest you start with one of the examples in Part I, since we won't explain *all* the details in this walkthrough.

In this post, we'll write a WhizzML script that automates a process to investigate Covariate Shift. To get an understanding of what we're trying to do, read this article first.[1]

Again, use the WhizzML reference guide[2] if you need help along the way.

Our goal is to write a function that:

- Given two datasets (one that represents the data used to train a predictive model, one that represents production data)
- Returns an indication of whether the distribution of data has changed.

As we read in the article, the indicator of change in our data distribution is called the `phi coefficient`. Our WhizzML script will return us this number, so lets name our base function `phi-coefficient`.

3.1 Phi-Coefficient

```
(define (phi-coefficient training-dataset production-dataset seed)
  (let (comb-data (combined-data training-dataset production-dataset)
        ids (split-dataset comb-data 0.8 seed)
        model (create-and-wait-model {"dataset" (ids 0)
                                      "objective_field" "Origin"})
        eval (model-evaluation model (ids 1)))
      (avg-phi eval))
```

What are we doing here?

To start, the function takes three arguments. The first two are ids for our training and production datasets, respectively. We call them `training-dataset` and `production-dataset`. The third argument, `seed`, is used to make our sampling deterministic. We'll talk about this later.

There's quite a bit going on in this function, but it's all broken into manageable pieces. First, we use `let` to set local variables. These local variables are the result of a few different functions which we will have to define.

The local variables are `comb-data`, `ids`, `model` and `eval`. After these are set, we can compute the phi coefficient with the function `avg-phi`.

[1] http://blog.BigML.com/2014/01/03/simple-machine-learning-to-detect-covariate-shift/
[2] https://bigml.com/whizzml

3.1.1 Comb-Data

`comb-data` is the result of `(combined-data training-dataset production-dataset)`

Here we're combining the two datasets into one big dataset. But before they are combined, we have to do a transformation on each dataset (add the "Origin" field). We'll talk about that transformation when we define `combined-data`.

The result of our `comb-data` dataset looks something like this:

```
|  field_1  |  field_2  |  ...  |  "Origin"  |
   123          124       ...    "Training"
   123          124       ...    "Production"
   123          124       ...    "Production"
   123          124       ...    "Training"
   ...          ...       ...       ...
```

3.1.2 Ids

Next, we have a variable called `ids`. This is a list of dataset-ids that is the result of

```
(split-dataset comb-data 0.8 seed)
```

What our `split-dataset` function does is takes the `comb-data` (one big dataset) and randomly splits it into two datasets. We split it so that we can train a predictive model with the larger portion of the split, and then evaluate its performance on the smaller part.

The `split-dataset` function returns something like this

`["dataset/83bf92b0b38gbgb" "dataset/83hf93gf012bg84b20"]`.

3.1.3 Model

`model` is a BigML predictive model resource. We are creating this model from the first element of our `ids` list: `"dataset" (head ids)`. The model is built to predict whether the value for the "Origin" field is "Training" or "Production". Thus, the "objective_field" is "Origin". `"objective_field" "Origin"`.

3.1.4 Eval

`eval` is a BigML evaluation resource. To create an evaluation, we need two arguments: a predictive model and a dataset we want to test the model against. Our model is stored in `model` and our dataset is the second element in the `ids` list, that is, the element in position 1: `(ids 1)`

3.1.5 Avg-Phi

We're done with the local variables, but what does the whole `phi-coefficient` function return - what's our end product?

```
(avg-phi eval)
```

That line gives us the average phi score for the evaluation we just created. A bunch of information is stored inside the `eval` data object that will be retrieved from BigML. But of course, we have to tell the function `avg-phi` *how to get* what we want! We'll save that for later.

. . .

So we have built our base function and understand its components. Now we have to go back and build the functions we haven't defined yet, specifically `comb-data`, `split-dataset`, `model-evaluation` and `avg-phi`. We'll start with `comb-data`.

3.2 Comb-Data

```
(define (combined-data training-dataset production-dataset)
  (create-and-wait-dataset {"origin_datasets" [(train-data training-dataset)
                                               (prod-data production-dataset)]}))
```

Again, this function combines two datasets. We tell BigML what datasets we want to combine using the `origin_datasets` parameter and passing it a list of dataset ids.

But what are `train-data` and `prod-data`?

Those are helper functions that add the "Origin" field we talked about.

- `train-data` adds the "Origin" field with the value "Training" in each row
- `prod-data` adds the "Origin" field with the value "Production" in each row

They are defined here:

```
(define (train-data dataset-id)
  (with-origin-field dataset-id  "Training"))

(define (prod-data dataset-id)
  (with-origin-field dataset-id "Production"))
```

Since we are doing pretty similar things in both functions, (adding an "Origin" field) we can separate that logic into its own function. Here it is:

```
(define (with-origin-field dataset-id value)
  (create-and-wait-dataset {"origin_dataset" dataset-id
                            "new_fields" [{"field" value
                                           "name"  "Origin"
                                           "label" "Origin"}]}))
```

In that function we are...

- Creating a new dataset from an existing one `"origin_dataset" dataset-id`
- Adding a new field `"new_fields" [...]`
- Giving the new field a column name and label `"name"  "Origin" "label" "Origin"`
- Setting the row's value `"field" value`.

The `value` will either be the string `"Production"` or `"Training"`. This string is passed in as an argument where `prod-data` and `train-data` are defined.

Nice. Now let's go over `split-dataset`.

3.3 Split-Dataset

```
(define (split-dataset dataset-id rate seed)
  (list (sample-dataset dataset-id rate false seed)
        (sample-dataset dataset-id rate true seed)))
```

- What are we splitting? `dataset-id`. The input dataset.

- How are we splitting it - 80%/20%? 90%/10%? We can do whatever we want. This is determined by `rate`.
- How are we going to shuffle our data before we split it? The `seed` determines this.

As you can see, we are sampling the same dataset twice. One sample will be used to build a predictive model, the other will be used to evaluate the predictive model.

`sample-dataset` is another function. Here it is below:

3.4 Sample-Dataset

```
(define (sample-dataset dataset-id rate oob seed)
  (create-and-wait-dataset {"sample_rate" rate
                            "origin_dataset" dataset-id
                            "out_of_bag" oob
                            "seed" seed}))
```

This function is what actually interacts with BigML. We create a new dataset, passing in the `rate`, the original dataset (`dataset-id`), whether it is `out_of_bag` or not (we'll go over this) and the `seed` used to determine how the original dataset was shuffled.

Here's a little diagram that will help explain how the `seed` and `out_of_bag` (oob) work.

```
                          (seed = "123")            (seed = "hi12")
                          (rate = 0.5)              (rate = 0.5)
-Original Dataset-      - New dataset -         - Another dataset -
| Id | field_1|         | Id | field_1|           | Id | field_1|
  1    "hello"            1    "hello"   x          1    "hello"   x
  2    "hi"               2    "hi"      x          2    "hi"      oob
  3    "ok"               3    "ok"      oob        3    "ok"      x
  4    "yeah"             4    "yeah"    oob        4    "yeah"    oob
```

So if `out_of_bag` is set to `true`, we grab the rows labeled "oob". Otherwise, we grab the ones marked "x" The `seed` just changes which rows we label "oob" and "x". The `seed` also enables this whole process to be deterministic. So if you run the `phi-coefficient` function with the same `seed` (and the same datasets), you'll get the same results!

Cool. That wraps up our `sample-dataset` and `split-dataset` functions. Next up, `model-evaluation`.

3.5 Model-Evaluation

```
(define (model-evaluation model-id dataset-id)
  (create-and-wait-evaluation {"model" model-id "dataset" dataset-id}))
```

We apologize if you were hoping for something more exciting. This function is just a wrapper for the method included with /whizzml, `create-and-wait-evaluation`. As you can see, we are simply creating an evaluation with a model and a dataset.

Our last function is. . .

3.6 Avg-Phi

```
(define (avg-phi ev-id)
  ((fetch ev-id) ["result" "model" "average_phi"]))
```

Pretty simple too!

We take the evaluation `ev-id` and fetch its data from BigML `(fetch ev-id)`. Then we access the `average_phi` attribute nested under "model" and "result".

The data object looks like this:

```
(fetch ev-id) ;; -> { "result" { "model" { "average_phi" 0.834 }}}
```

...

And there we have it. A WhizzML script that helps predict covariate shift.

3.6.1 All Together:

```
(define (with-origin-field dataset-id value)
  (create-and-wait-dataset {"origin_dataset" dataset-id
                            "new_fields" [{"field" value
                                           "name"  "Origin"
                                           "label" "Origin"}]}))

(define (train-data dataset-id)
  (with-origin-field dataset-id  "Training"))

(define (prod-data dataset-id)
  (with-origin-field dataset-id "Production"))

(define (combined-data training-dataset production-dataset)
  (create-and-wait-dataset {"origin_datasets" [(train-data training-dataset)
                                               (prod-data production-dataset)]}))

(define (sample-dataset dataset-id rate oob seed)
  (create-and-wait-dataset {"sample_rate" rate
                            "origin_dataset"  dataset-id
                            "out_of_bag" oob
                            "seed" seed}))

(define (split-dataset dataset-id rate seed)
  (list (sample-dataset dataset-id rate false seed)
        (sample-dataset dataset-id rate true seed)))

(define (model-evaluation model-id dataset-id)
  (create-and-wait-evaluation {"model" model-id "dataset" dataset-id}))

(define (avg-phi ev-id)
  ((fetch ev-id) ["result" "model" "average_phi"]))

(define (phi-measure training-dataset production-dataset seed)
  (let (comb-data (combined-data training-dataset production-dataset)
        ids (split-dataset comb-data 0.8 seed)
        model (create-and-wait-model {"dataset" (ids 0)
                                      "objective_field" "Origin"})
        ev-id (model-evaluation model (ids 1)))
      (avg-phi ev-id)))
```

We can run our function like this:

```
(phi-measure "datast/ei9202i390203" "dataset/s93999303f09" "test-run-1") -> 0.82
(phi-measure "datast/ei9202i390203" "dataset/s93999303f09" "test-run-2") -> -0.4
```

But...

As we read in the article, it is best to do this process several times and look at the average of the results. How could we add some more code to to do this programmatically?

Here's one implementation.

3.7 Multi-Phis

```
(define (multi-phis n training-dataset production-dataset)
  (loop (seeds (range 0 n) out [])
    (if (= [] seeds)
      {"list" out
       "average" (/ (reduce + 0 out) (count out))}
      (recur (tail seeds)
             (append out (phi-coefficient training-dataset
                                          production-dataset
                                          (str "test-" (head seeds))))))))
```

Again, we are giving this function our `training-dataset` and `production-dataset`. But we are also passing in `n`, which is the number of phi-coefficients we want to calculate.

As you can see, we are defining a `loop`.

Within this loop, we set some variables.

- `seeds`, we give the default (starting) value of `(range 0 n)`
  - If we pass in 4 for the value of `n` then the initial value of `seeds` = `[0 1 2 3]`
- `out` is our output. We will add the result of a `phi-coefficient` run each time through the loop.
  - Initially, `out` = `[]`

We also define the end-scenario.

- If `seeds` is empty, then we return a map with the values `list` and `average`. (we'll explain these in a bit)
- If `seeds` is not empty, we go back to the loop, but define values for `seeds` and `out`.
- `seeds` = `(tail seeds)`. This grabs everything but the first element of `seeds`
  - So the first time through, it might be `[0 1 2 3]`, then it will be `[1 2 3]`, then `[2 3]`...
- `out` = `(append out (phi-coefficient ...))` We take the result of our `phi-coefficient` function and add it to the `out` list.
  - First time through, it's `[]`, then `[-0.0838]`, then `[-0.0838, 0.1240]` ...

The `seed` we will use for each of these `phi-coefficient` runs will be `"test-0"`, `"test-1"`, `"test-2"` etc. Thats what `(str "test-" (head seeds))` is doing - joining the string `"test-"` with the first element of the `seeds` list.

The last thing we should discuss is the end-case return value:

```
{"list" out
 "average" (/ (reduce + 0 out) (count out))}
```

The value of "list" (`out`) is just the list of phi-coefficient values from each run. The "average" is... Yep. The average of all the runs. `reduce` adds up the elements. `count` counts the number of elements. `/` divides the first by the second.

You got it! The average of many phi-coefficients between two datasets, to help predict covariate shift.

Example run[3]:

```
(multi-phis 3 "dataset/56c3af3f7e0a8d6cca01292e" "dataset/574ef59546522f61f2000444")
;; -> { :list [0.03824, -0.10747, -0.08768], :average -0.05230 }
```

Cool

Take a second and think about what you can accomplish now in a few clicks with this WhizzML Script.

1. Make a bunch of predictive models
2. Evaluate their performances
3. Get the knowledge of whether your data characteristics have changed

Sweet!

And even more powerful... The **knowledge to automate your own processes!**

[3] Since we used the same dataset for the Training and Production data, it guesses the wrong value for the "Origin" nearly every time! That's why the phi-coefficient values are all close to -1.

CHAPTER 4

Best-K

BigML has an implementation of G-means clustering built into the cluster API.[1] G-means clustering is an enhancement to K-means clustering that seeks to find the optimal value of K under the assumption that the neighborhood of points around the centroid of a cluster should have a Gaussian distribution in a certain sense.[2] In our API, G-means clustering is invoked by omitting the number K of centroids for K-means clustering, *e.g*:

```
(create-and-wait-cluster {"dataset" "dataset/57311df8b95b394f4e000111"
                          "critical_value" 5
                          "name" "G-means Clustering Example"})
```

Assuming a priori that clusters in a dataset conform to the assumptions of the G-means algorithm may be inappropriate. In addition, the G-means method for determining K incrementally fissions clusters and adjusts the resulting clusters appropriately, so that in many cases not every value of K is considered. This proclivity of the algorithm to split a cluster into two clusters is determined by the `critical_value` parameter, but frequently there aren't principled ways to select its value.

Fortunately, using WhizzML we can implement an alternative to G-means for determining K based on the Pham-Dimov-Nguyen algorithm..[3] Pham, Dimov, and Nguyen define a measure of concentration $f(K)$ on a K-means clustering and use that as evaluation function to determine the best K. This tutorial presents an example implementation of the PDN-based algorithm that finds the best number K of centroids for K-means clustering in an arbitrary range of $K_{\min}$ to $K_{\max}$.

The full code for this tutorial is available in our whizzml examples repository[4] on Github.

4.1 Code Overview

We first present a brief overview of the WhizzML functions in this package that implement the Pham-Dinov-Nguyen approach to estimating the best number of centroids K for a K-means clustering. We then review a few brief example functions using the top-level functions in the package.

```
(define (generate-clusters dataset args k-min k-max)
  (let (dname ((fetch dataset) "name")
        fargs (lambda(k)
                (assoc args "dataset" dataset
                            "k" k
                            "name" (str dname " - cluster (k=" k ")")))
        clist (map fargs (range k-min (+ 1 k-max)))
```

[1] https://bigml.com/developers/clusters
[2] https://blog.bigml.com/2015/02/24/divining-the-k-in-k-means-clustering/
[3] http://www.ee.columbia.edu/~dpwe/papers/PhamDN05-kmeans.pdf
[4] https://github.com/whizzml/examples/tree/master/best-k

```
        ids (create* "cluster" clist))
    (map fetch (wait* ids))))
```

Inputs:

- `dataset`: (`string`) Dataset identifier of the dataset to be clustered
- `args`: (`map`) Arguments for the cluster operation
- `k-min`: (`number`) Minimum value of K
- `k-max`: (`number`) Maximum value of K

Output: (`list`) Cluster metadata for created clusters

We begin with a function `generate-clusters` to create a collection of K-means clusterings for an arbitrary range `k-min` $\leq K \leq$ `k-max` of centroids. The input map `args` for the WhizzML cluster function is expanded to `fargs` with the input `dataset`, the number K of centroids, and the name for each cluster instance. After setting up the list of arguments for generating each cluster in the collection, the routine uses the common idiom `create*`, `wait*` to generate the returned set of candidate BigML Cluster objects in parallel.[5]

```
(define (extract-eval-data cluster)
  (let (id (cluster "resource")
        k (cluster "k")
        n (count (cluster "input_fields"))
        within_ss (cluster ["clusters" "within_ss"])
        total_ss (cluster ["clusters" "total_ss"]))
    {"id" id "k" k "n" n "within_ss" within_ss "total_ss" total_ss}))
```

Inputs:

- `cluster`: (`string`) Cluster identifier of the cluster

Output: (`map`) Cluster metadata used to compute the evaluation function $f(K)$

To simplify the implementation of the PDN algorithm, we next define a helper function to extract certain metadata items from the full metadata for a cluster. The metadata returned by `extract-eval-data` are just the items needed to compute the PDN evaluation function $f(K)$. These include the number K of centroids in the K-means clustering, the number of covariates n considered in the K-means computation, the total sum-squared distance between the items in the cluster and the cluster centroid for all clusters S_k (`within_ss`), and finally S_1 available in the metadata for every cluster (`total_ss`).

We could have included this helper function in the `generate-clusters` function. We chose here to define it separately for illustrative purposes. Users may find other application-specific alternatives.

```
(define (alpha-func n)
  (let (alpha_2 (- 1 (/ 3 (* 4 n)))
        w (/ 5 6))
    (lambda (k)
      (if (<= k 2)
        alpha_2
        (+ (* (pow w (- k 2)) alpha_2) (- 1 (pow w (- k 2)))))))))
```

Inputs:

- `n`: (`number`) Number of covariates

Output: (`function`) Weighting function $\alpha(K)$

[5] The reader might have observed this routine illustrates how WhizzML is not a pure functional language. The primary intent of WhizzML is to orchestrate BigML operations through side effects such as, in this case, creating BigML clusters from a BigML dataset.

As discussed in more detail below, the Pham-Dimov-Nguyen algorithm is based on an evaluation function $f(K)$ that includes a weighting $\alpha(K)$ function parameterized on the number of covariates n. The weighting function in the Pham-Dimov-Nguyen paper is in recursive form. This factory function returns the closed-form equivalent:

$$\alpha(K) = \begin{cases} 1 - 3/4n & k = 2 \\ (5/6)^{k-2}\alpha(2) + \left[1 - (5/6)^{k-2}\right] & k > 2 \end{cases}$$

In the same manner as we do next with `evaluation-func`, we can implement $\alpha(K)$ as the partial application of a function $\alpha(K, n)$ for n. We realize this partial function application as a factory function `apply-func`. This factory function defines a closure that in turn includes n and returns an anonymous function `lambda(k)` as $\alpha(K)$.

```
(define (evaluation-func n)
  (let (fa (alpha-func n))
    (lambda (k sk skm)
      (if (or (<= k 1) (not skm) (zero? skm))
        1
        (/ sk (* (fa k) skm))))))
```

Inputs:

- `n`: (`number`) Number of covariates

Output: (`function`) Weighting function $\alpha(K)$

The Pham-Dimov-Nguyen approach to finding the best K has at its core an evaluation function $f(K)$. The version in the Pham-Dimov-Nguyen paper is a function of a single argument K that internally includes S_K and S_{K-1} (the `within_ss` field of the cluster metadata map returned by `extract-eval-data`). The form returned by this factory function has the S_K and S_{K-1} values as arguments.

$$f(K, S_K, S_{K-1}) = \begin{cases} 1 & k = 1 \text{ or } S_{K-1} = 0 \text{ or } S_{K-1} \text{ undefined} \\ S_K/[\alpha(K)S_{K-1}] & \text{otherwise} \end{cases}$$

Following the conventions of functional programming languages we can implement $f(K, S_K, S_{K-1})$ as the partial application of a function $f(K, S_K, S_{K-1}, n)$ for n.

This factory function `evaluation-func` defines a closure that in turn includes n and returns an anonymous function `lambda(k sk skm)` as $f(K, S_K, S_{K-1})$. Note also that the weighting function $\alpha(K)$ in the PDN evaluation $f(K)$ could have been a subfunction inside this `evaluation-func`. As discussed previously, for illustrative purposes we instead have implemented it as the returned result of a separate factory function `alpha-func`.

```
(define (evaluate-clusters clusters)
  (let (cmdata (map extract-eval-data clusters)
        n ((cmdata 0) "n")
        fe (evaluation-func n))
    (loop (in cmdata
           out []
           ckz {})
      (if (= [] in)
        out
        (let (ck (head in)
              ckr (tail in)
              k (ck "k")
              within_ss (ck "within_ss")
              within_ssz (if (<= k 2) (ck "total_ss") (ckz "within_ss"))
```

```
            cko (assoc ck "fk" (fe k within_ss within_ssz)))
          (recur ckr (append out cko) ck))))))
```

Inputs:

- `clusters`: (`list`) Cluster metadata maps ordered by K

Output: (`list`) Sequence of maps that have the field `fk` with the value $f(K, S_K, S_{K-1})$ added

Having defined a number of component functions, we pull them together in `evaluate-clusters`. This function applies the Pham-Dimov-Nguyen evaluation function $f(K)$ to a list that ranges over K of the K-means clusters for a dataset. The result is a list over K of items that include the value of $f(K)$.

In more detail, `evaluate-clusters` applies the evaluation function $f(K, S_K, S_{K-1})$ returned by `evaluation-func` as `fe` to the cluster metadata returned by `extract-eval-data` as `cmdata`. The body of the function is a `loop` that iterates over the `in` list of metadata maps returned by `extract-eval-data` and sequentially builds the `out` list of metadata maps. Each map in the `out` list is the source map in the `list` list augmented with the value $f(K, S_K, S_{K-1})$ as `fk` of the evaluation function `fe` applied to the data values K and S_K (`within_ss`) from the input map. In addition, the input cluster metadata map `ck` to an iteration of the loop is passed back into the next iteration as the source `ckz` for the value of S_{K-1}.

```
(define (clean-clusters evaluations cluster-id logf)
  (for (x evaluations)
    (let (id (x "id")
          _ (if logf (log-info "Testing for deletion " id " " cluster-id)))
      (if (!= id cluster-id)
        (prog (delete id)
              (if logf (log-info "Deleted " id))))))
  cluster-id)
```

Inputs:

- `evaluations`: (`list`) Sequence of maps of evaulation results for the clusters
- `cluster-id`: (`string`) Cluster to save (not delete)
- `logf`: (`boolean`) Flag to enable logging

Output: (`string`) Returns the `cluster-id` supplied as an input.

Before we define the final, top-level functions we define two more helper functions. The first is this straightforward helper function `clean-clusters` that deletes the BigML cluster objects created as intermediate computation results, except for the cluster specified by `cluster-id`.

```
(define (best-cluster dataset args k)
  (let (dname ((fetch dataset) "name")
        ckargs (assoc args "dataset" dataset
                           "k" k
                           "name" (str dname " - cluster (k=" k ")")))
    (create-and-wait-cluster ckargs)))
```

Inputs:

- `dataset`: (`string`) Identifier of dataset to be processed with the cluster operation.
- `cluster-arg`: (`map`) Arguments for cluster function
- `k`: (`number`) Number of centroids for cluster operation

Output: (`string`) Returns the `cluster-id` of the created cluster.

The second helper function `best-cluster` is also required by some of our top-level functions. It performs a single K-means clustering with the specified number K of centroids. The input map `args` for the

WhizzML cluster function is expanded to `ckargs` with the identifier of the input `dataset`, the number K of centroids `k`, and the `name` for the cluster instance.

```
(define (evaluate-k-means dataset args k-min k-max clean logf)
  (let (clusters (generate-clusters dataset args k-min k-max)
        evaluations (evaluate-clusters clusters))
    (if clean
      (clean-clusters evaluations "" logf))
    evaluations))
```

Inputs:

- `dataset`: (`string`) identifier of dataset to be processed with the cluster operation.
- `args`: (`map`) Arguments for cluster function
- `k-min`: (`number`) Minimum value of K
- `k-max`: (`number`) Maximum value of K
- `clean`: (`boolean`) Flag to delete all but the optimal cluster
- `logf`: (`boolean`) Flag to enable logging

Output: (`list`) Pham-Dimov-Nguyen evaluations of the clusters.

This function combines the `generate-clusters` and `evaluate-clusters` functions to create a list of K-means clusters with a range of centroids and a list of metadata maps that include the Pham-Dimov-Nguyen evaluation function $f(K)$ values for those clusters.

The `args` argument is a map that one can use to optionally specify all of the parameters for the K-means cluster function except the `dataset`, the number of centroids `k`, and the cluster `name` parameters. (See the the BigML developer documentation[6] for details.) The `dataset`, `args`, and `k-min` and `k-max`, are passed directly to the `generate-clusters` function. The list of cluster metadata for the clusters it creates is passed directly to the `evaluate-clusters` function.

This function can be used as top-level function to just return the list of Pham-Dimov-Nguyen evaluation functions $f(K)$ result over the specified number of centroids `k-min` to `k-max` (inclusive). When called as top-level function, the `clean` parameter can be specified as `true` to automatically delete the BigML cluster objects created after the PDN evaluations are computed. Setting the boolean parameter `logf` to `true` causes generation of log data.

```
(define (best-k-means dataset args k-min k-max best-args clean logf)
  (let (evaluations (evaluate-k-means dataset args k-min k-max false logf)
        _ (if logf (log-info "Evaluations " evaluations))
        besteval (min-key (lambda (x) (x "fk")) evaluations)
        _ (if logf (log-info "Best " besteval))
        cluster-id (if (= args best-args)
                     (besteval "id")
                     (best-cluster dataset best-args (besteval "k"))))
    (if clean
      (clean-clusters evaluations cluster-id logf))
    cluster-id))
```

Inputs:

- `dataset`: (`string`) identifier of dataset to be processed with the cluster operation.
- `cluster-arg`: (`map`) Arguments for cluster function
- `k-min`: (`number`) Minimum value of K

[6] https://bigml.com/developers/clusters#cl_cluster_arguments

- `k-max`: (`number`) Maximum value of K
- `clean`: (`boolean`) Flag to delete all but the optimal cluster
- `logf`: (`boolean`) Flag to enable logging

Output: (`string`) Cluster identifier of best K-means cluster as determined by Pham-Dimov-Nguyen evaulation function.

This top-level function uses the Pham-Dimov-Nguyen algorithm to create the best K-means clustering and returns the identifier of the BigML cluster object for the number of centroids K that results in the smallest (best) value of the PDN evaluation function $f(K)$.

The `clusters-args` and `best-args` parameters are maps that one can use to optionally specify all of the parameters for the cluster function except the `dataset`, the number of centroids `k`, and the cluster `name` parameters (See the the BigML developer documentation[7] for details). `args` is used in the search phase to find the best K. `best-args` allows one to specify different args for the final stage of clustering with the best K. In particular, one might do clustering on a subset of the `dataset` during the search phase to save time and other resources, then do the best clustering on the full `dataset`. If `best-args` matches `args`, the result for the best K generated with `args` during the search phase is returned by `best-k-means`. If `best-args` differs from `args`, the `dataset` is re-clustered with the best K and the identifier of that cluster is returned by `best-k-means`.

This function is called by `best-batchcentroid` to do the actual clustering. It can also be called directly if one only needs the best WhizzML cluster object and not the full WhizzML batchcentroid and annotated dataset.

```
(define (best-batchcentroid dataset args k-min k-max best-args clean logf)
  (let (cluster-id (best-k-means dataset args k-min k-max best-args clean logf)
        batchcentroid-id (create-and-wait-batchcentroid {"cluster" cluster-id
                                                         "dataset" dataset
                                                         "output_dataset" true
                                                         "all_fields" true}))
    batchcentroid-id))
```

Inputs:

- `dataset`: (`string`) Identifier of dataset to be processed with the cluster operation.
- `cluster-arg`: (`map`) Arguments for cluster function
- `k-min`: (`number`) Minimum value of K
- `k-max`: (`number`) Maximum value of K
- `clean`: (`boolean`) Flag to delete all but the optimal cluster
- `logf`: (`boolean`) Flag to enable logging

Output: (`string`) Identifier of created batchcentroid

This final top-level routine first uses the `best-k-means` function to generate a best K-means clustering determined by the Pham-Dimov-Nguyen evaluation function $f(K)$. It then creates a BigML batchcentroid object and BigML dataset annotated with clusters numbers in the best K-means clustering of the supplied dataset.

4.2 Examples

We conclude with a few brief examples of how to use the top-level functions and some additional application suggestions.

[7] https://bigml.com/developers/clusters#cl_cluster_arguments

Using the Top-Level Functions The main top-level function can be called to generate the best BigML batchcentroid and an annotated BigML dataset for an input `dataset` with the identifier `ds-id` as:

```
(define batchcentroid-id (best-batchcentroid ds-id {} 1 10 {} false false))
```

If only the best K-means BigML cluster object is required, perhaps for evaluating different ranges of `k-min` to `k-max`, one can use the top-level function:

```
(define cluster-id (best-k-means ds-id {} 1 10 {} false false))
```

Finally, if one only seeks the list of Pham-Dimov-Nguyen evaluation function $f(K)$ results for the K-means clusterings for `k-min` to `k-max`, one can use the top-level function:

```
(define evaluations (evaluate-k-means ds-id {} 1 10 false false))
```

Specifying `clean` as `true` causes these top-level functions to delete the intermediate BigML datasets created during the computation. Other information generated during execution of function can be logged by specifying `logf` as `true`.

Using `args` and `best-args` The input `args` can be used to specify a custom map of arguments for the BigML cluster function that generates the collection of clusters for `k-min` to `k-max` in the evaluation phase. Similarly, the input `best-args` in the `best-k-means` and `best-batchcentroid` functions can be used to specify a custom map of arguments for the K-means cluster operation with the best number K of centroids.

As an example, suppose we have a dataset that has two fields `depvar1` and `depvar2` that we consider to be dependent variables predicted from the remaining fields of the dataset. We'd like to discover the best number of clusters when only the remaining fields are considered by the K-means algorithm. In addition, we believe we can get a reasonable estimate for the best K if we only use 20% of the dataset in the evaluation phase. We can limit the K-means cluster computations according to these constraints by specifying

```
(define ds-id "dataset/...")
(define cfargs {"excluded_fields" ["depvar1" "depvar2"]})
(define cargs (assoc cfargs "seed" "test seed" "out_of_bag" false "sample_rate" 0.2))

(define evaluations (evaluate-k-means ds-id cargs 1 10 false false))
(define cluster-id (best-k-means ds-id cargs 1 10 cfargs false false))
(define batchcentroid-id (best-batchcentroid ds-id cargs 1 10 cfargs false false))
```

Segmented Searches for the Best K In some cases we might have a first guess for the best K. When we run the evaluations over an initial range `k-min` to `k-max` we find that the values of $f(K)$ over that range do not exhibit a clear and unambiguous minimum. Rather than re-run the evaluations over the initial range `k-min` to `k-max`, we can simply rerun it over additional subranges.

For instance, we can run:

```
(define evaluations (evaluate-k-means ds-id cargs 1 10 false false))
(define evaluations (evaluate-k-means ds-id cargs 10 20 false false))
(define evaluations (evaluate-k-means ds-id cargs 20 30 false false))
```

to cover the total range `k-min = 1` to `k-min = 30`.

There is one detail to note here. The initial value $f(K_{\min})$ for $K_{\min} > 1$ is an approximation. By overlapping the ranges as shown in the example, the initial value for any $K_{\min} > 1$ can be discarded. In this case, $f(K)$ over the full range $1 \leq K \leq 30$ can be assembled by composing $f(K)$ for $1 \leq K \leq 10$ from the first run, $11 \leq K \leq 20$ from the second run, and $21 \leq K \leq 30$ from the third run.

Part III

Advanced

The purpose of the Advanced tutorials is to demonstrate porting high-level Machine Learning algorithms to the WhizzML language. Writing the scripts in this section requires a thorough understanding of Machine Learning concepts. To this end, lower level WhizzML concepts will not be explained in detail here. For more basic instruction regarding the WhizzML language, refer to the Part I or Part II tutorials, or the WhizzML Language Reference guide.

CHAPTER 5

Gradient Boosting

Gradient tree boosting has become one of the more popular algorithms in the machine learning world, thanks in part to a number of open-source software packages as well as some high profile public successes.

The algorithm is, as bagged classifiers and random forests, an ensemble of weak learners, combined to make a strong one. The primary difference with boosting is that each successive weak model is, in a sense, trained on the mistakes of the previously learned models. With gradient boosting, a gradient is calculated with respect to the current model collection, and each model represents a gradient "step" in that direction.

Because the gradient steps are expressed as trees, one way to code this algorithm is using WhizzML to specify and string together the various steps in the algorithm, and allow BigML's computing infrastructure to handle the heavy lifting of calculation over all points in the dataset and the individual modeling steps.

The full code for this tutorial is available in our whizzml examples repository[1] on Github.

Advanced Tutorial Alert! This tutorial is pretty advanced. It assumes a significant level of comfort with WhizzML and with a bit of high-ish level mathematics. If you feel like you're in over your head, you can always head back to the Part I or the Part II sections to get your feet on firmer ground before you continue here.

It might also be helpful to have some of the other WhizzML reference material[2] on hand to help you as you follow along.

5.1 Algorithm Overview

As mentioned above, gradient tree boosting is an iterative algorithm that learns an ensemble of decision trees. Each tree is trained on the mistakes of all previous trees, and so each tree represents a step towards more correct predictions.

Gradient tree boosting is a gradient descent algorithm in the most general sense of that term, in which an objective function is minimized via gradient descent. Normally, however, the gradient steps to be taken are computed analytically from a parameterized model by differentiating the objective with respect to these parameters. In gradient tree boosting, we have only the predictions of the model on the training data as our expression of the current model parameters. Thus, we compute the gradient at each of the training data points and use decision trees to craft an approximate representation.

More specifically, suppose our current predictions for the training data are represented as an $n \times k$ matrix of probabilities, where n is the number of points and k is the number of classes. These probabilities are expressions of the model parameters. Without going into the actual objective function used, it turns out that a reasonable approximation for the gradient at the training data points is simply $\hat{y} - p_i$, where $\hat{y}$ is 1 if the true class is class $i \in 1 \ldots k$ and 0 otherwise.

[1] https://github.com/whizzml/examples/tree/master/gradient-boosting
[2] https://bigml.com/whizzml

We then train a model (rather, k models, one for each class) to approximate this difference at each of the training points, then uses the model to predict the value on the training data. Why is this necessary, as we know the value of the objective at these points already (that's how we trained the model!)? First, as it's important that we don't overfit the training data, we'll learn a "shallow" tree so the predictions will be different from the true values, and should be a good proxy for the predictions we'd see on new data.

With the predictions for this gradient "step" in hand, we can sum them onto our current running sum of gradient steps, then use the *softmax transform* to turn these sums into probabilities. We use the softmax function here for more than just convenience; the objective function that we glossed over two paragraphs ago has this softmax operation as an important part of it, so not using it here would be mathematically invalid.

With these probabilities in hand, we can recompute the gradient at the training points and iterate the algorithm. The algorithm stops generally if you reach some predetermined number of iterations or cease to make improvement for a while (we'll do the later here).

You're left finally with an $m \times k$ matrix of models, where m is the number of gradient steps you took and k is the number of classes. To predict probabilities for a new point, predict a score for that point with each model, sum these scores "down the columns" so you're left with k sums, one for each class, then apply the softmax transform to get your per-class probabilities.

Yes, this is a lot of information. And we're ignoring quite a bit; we're not going to mess with learning rates, shrinkage, or regularization. But you could if you wanted! Check out these slides[3] for more information if you want to dig deeper.

All right, enough exposition. Let's write some code.

5.2 Helper Functions

Before we get into the guts of the algorithm, we're going to introduce a few helper functions.

```
;; A constant added to the generated field names to let us know that
;; we generated them
(define boost-id "__bmlboost")

;; The names of the fields contain ground truth - if there are k
;; classes, this is k columns, one for each class.  If the true class
;; for a given point is the nth class, the value in column in for that
;; point is 1, else it is zero.
(define (truth-names nclasses)
  (map (lambda (i) (str boost-id "_truth_" i)) (range nclasses)))

;; For each of the "names" classes below, we are generating field
;; names, one for each class, at each iteration of the algorithm.

;; This generates a unique field name given a prefix `name` and an
;; iteration number.
(define (field-names nclasses iteration name)
  (map (lambda (i) (str boost-id "_" name "_" i "_iter_" iteration))
       (range nclasses)))

;; The names for the fields containing the total scores (the running
;; sum of all gradient steps) at iteration `iteration`
(define (sum-names nclasses iteration)
  (field-names nclasses iteration "sum"))

;; The names for the fields containing the scores at iteration `iteration`
```

[3] https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf

```
(define (pred-names nclasses iteration)
  (field-names nclasses iteration "prediction"))

;; Field names for the softmax probabilities at iteration `iteration`
(define (softmax-names nclasses iteration)
  (field-names nclasses iteration "softmax"))

;; The field name for the gradients (the objective for each class) at
;; each iteration
(define (grad-names nclasses iteration)
  (field-names nclasses iteration "gradient"))
```

As we make our way through the algorithm, we're going to create a lot of new columns in our dataset. We'll create new columns for the gradients at each step, the running sums at each step, and the predictions of the gradient trees at each step. These functions will return unique names for each type of column; one for each class at any given iteration of the algorithm

```
;; Helper methods to add fields to the given dataset using flatline
;; expressions
(define (make-fields names exprs)
  (let (make-field (lambda (i) {"name" (names i) "field" (exprs i)}))
    (map make-field (range (min (count exprs) (count names))))))

(define (add-fields dataset new-fields input-ids)
  (let (req {"origin_dataset" dataset "new_fields" new-fields})
    (if (empty? input-ids)
      (create-and-wait-dataset req)
      (create-and-wait-dataset (assoc req "input_fields" input-ids)))))
```

Here's how we'll actually create all of those new columns. The top function `make-fields` takes a list of column names and list of flatline expressions. It then creates a list of maps with the names and expressions in each one. The bottom function takes this list, puts into a BigML resource request and creates a copy of the argument dataset with the new columns appended. If `input-ids` is specified, it retains only those fields from the original dataset.

Wait, you haven't heard of flatline? That's BigML's DSL for dataset transformation. As you'll see, using WhizzML to compose flatline expressions that transform your data allows you to do a ton of interesting things. You can get more familiar with flatline by perusing the user's guide here.[4]

```
;; Get the original input fields from the dataset, to make sure we use
;; the same fields to learn at each iteration.
(define (get-inputs fields)
  (let (not-generated? (lambda (astr) (not (contains-string? boost-id astr)))
        is-input? (lambda (fid) (not-generated? ((fields fid) "name"))))
    (filter is-input? (keys fields))))

;; Get the objective field ids for the given iteration
(define (get-objectives fields nclasses iteration)
  (let (gnames (grad-names nclasses iteration))
    (map (lambda (name) (id-from-fields fields name)) gnames)))

;; Get the total number of classes for the problem from the field
;; descriptor
(define (get-num-classes dataset obj-id)
  (let (obj ((get-fields dataset) obj-id))
```

[4] https://github.com/bigmlcom/flatline/blob/master/user-manual.md

```
    (count (obj ["summary" "categories"])))))
```

Finally, a few miscellaneous helpers: `get-num-classes` does what it says on the box, gets the number of classes in the objective field. `get-inputs` gets the original input fields for the dataset, a handy thing to know when you're adding a whole bunch of columns at each iteration, `get-objectives` gets the field ids of the objective fields (one for each class) with the gradient columns at a given iteration.

5.3 Computing the Gradients Given Probabilities

Recall that the first thing we have to do is compute the gradient at each of the training data points. We're going to use flatline to subtract the currently predicted probability for each class from the true objective value, then put that value in a new set of columns (again, one for each class).

This requires us to have a predicted probability for each point. During the first iteration, we're just going to assume all classes have equal probability.

```
;; Compute the gradient given the ground truth fields and the current
;; probabilities
(define (compute-gradient dataset nclasses iteration)
  (let (next-names (grad-names nclasses iteration)
        preds (if (> iteration 0)
                  (map (lambda (n) (flatline "(f {{n}})"))
                       (softmax-names nclasses iteration))
                  (repeat nclasses (str (/ 1 nclasses))))
        tns (truth-names nclasses)
        fexp (lambda (idx)
               (let (actual (tns idx)
                     predicted (preds idx))
                 (flatline "(- (f {{actual}}) {predicted})")))
        new-fields (make-fields next-names (map fexp (range nclasses))))
    (add-fields dataset new-fields [])))
```

We first get the names of the columns where we're going to put the gradient values in `next-names`. Then we get the currently predicted probabilities for each class. If it's an iteration other than the first, we compose a flatline expression that pulls out the value from the appropriate column, given by `softmax-names`.

We can then write a little internal function, `fexp`, which subtracts the two columns values. Of course, we have to `(map fexp (range nclasses))` so that we get an expression for each class, then use `new-fields` and `add-fields` to add them to the dataset.

5.4 Learning the Gradient Models

Now that we have our collection of gradient columns, we're going to learn a tree to represent each one.

```
;; Learn a set of trees over the objective fields, one for each class
(define (learn-trees dataset nclasses iteration)
  (let (fs (get-fields dataset)
        iids (get-inputs fs)
        oids (get-objectives fs nclasses iteration)
        req {"dataset" dataset "input_fields" iids}
        create (lambda (oid) (create-model (assoc req "objective_field" oid)))
        ids (map create oids)
        _ (wait-forever* ids))
    ids))
```

Here we get the input and objectives for each model (remember, `get-inputs` returns the original input fields for the data, minus the fields we've generated ourselves). The input fields will always be the same, but we will learn one model per objective field (that is, one per class).

We do this with the inline function `create`, which takes the template `req` for the model request and adds an objective field. We then `(map create oids)` to create one model for each objective in `oids` and wait for them to complete.

5.5 Scoring Instances with the Models

The next step is to use our learned models to predict gradient values for the training data.

```
;; Predict the value of the gradient for all points in the dataset
;; Need to predict one at a time so we can preserve all fields
(define (batch-predict dataset iteration mod-ids)
  (let (pnames (pred-names (count mod-ids) iteration))
    (loop (last-ds dataset mids mod-ids names pnames)
      (if (empty? mids)
        last-ds
        (let (req {"all_fields" true
                   "output_dataset" true
                   "model" (head mids)
                   "dataset" last-ds
                   "prediction_name" (head names)}
              bp (create-and-wait-batchprediction req)
              new-ds ((fetch bp) "output_dataset_resource")
              _ (wait-forever new-ds))
              (recur new-ds (tail mids) (tail names)))))))
```

This code works by first getting names for the columns where we'll put the predictions (`pnames`). We then loop over these names and the corresponding model identifiers, creating a `batchprediction` for each one. We'll do these serially, as we want to preserve predictions from previous models when we make a new one, so the dataset created by the first `batchprediction` will provide the input fields for the second, and so on for each model.

Note that we have to `create-and-wait-batchprediction` and later wait separately for the created dataset; the readiness of the batch prediction does not imply readiness for the output dataset resource.

5.6 Summing the Model Scores Into The Current Model

Now we have to sum our predicted scores into our running totals. We'll use flatline again.

```
;; Sum the last set of predictions with the current set of sums to get
;; new scores
(define (create-sums dataset nclasses iteration)
  (let (this-preds (pred-names nclasses iteration)
        this-sums (sum-names nclasses iteration)
        last-sums (if (> iteration 1) (sum-names nclasses (- iteration 1)) [])
        fexp (lambda (idx)
               (let (this-pred (this-preds idx))
                 (if (empty? last-sums)
                   (flatline "(f {{this-pred}})")
                   (let (last-sum (last-sums idx))
                     (flatline "(+ (f {{this-pred}}) (f {{last-sum}}))")))))
        new-fields (make-fields this-sums (map fexp (range nclasses))))
    (add-fields dataset new-fields [])))
```

This looks a lot like the function we used to compute the gradient and follows the same general pattern. First we pull out the names "source" and "destination" columns for the flatline operation (in this case, `last-sums` and `this-preds`, and `this-sums` respectively.

We then create a function `fexp` that will take a class index i, pick out the ith name from each of the aforementioned lists, and compose the flatline expression that adds the two columns. Of course, if we're on the first iteration of the algorithm (`(empty? last-sums)`) then the current sum is just the current prediction.

We can then `(map fexp (range classes))` to do this for each class, resulting in a new running sum for each class. Finally, we use `make-fields` and `add-fields` to add the list of new columns to the dataset.

5.7 Computing Probabilities From Scores

Essentially the last step in the algorithm is to turn the set of running sums into probabilities that can feed into the next iteration of the algorithm. For this we'll use the *softmax transform* to massage the values into a proper distribution. What is it? Suppose you've got k classes and a score for each class $s_1, s_2, \ldots, s_k$. We define the probability p_i for class i as:

$$p_i = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

so that the probability vector is the vector of scores exponentiated, then normalized.

```
;; Create the softmax probabilities from the given scores
(define (create-softmax-probs dataset nclasses iteration)
  (let (this-sums (sum-names nclasses iteration)
        this-softmaxs (softmax-names nclasses iteration)
        fl-exp (lambda (name) (flatline "(exp (f {{name}}))"))
        exp-sum (str "(+ " (join " " (map fl-exp this-sums)) ")")
        fexp (lambda (name) (str "(/ " (fl-exp name) " " exp-sum  ")"))
        new-fields (make-fields this-softmaxs (map fexp this-sums)))
    (add-fields dataset new-fields [])))
```

Again, we rely on flatline to do the heavy lifting. We get the source and destination columns as before in `this-sums` and `this-softmaxs`. We then compose a flatline expression equivalent to the above equation. Note that we do it by parts here for convenience, first defining an "exponentiator" in `fl-exp`, then the denominator in `exp-sum`, and finally the closure that will make the final expression for each column in `fexp`. One more invocation of `new-fields` and `add-fields` and we have our new probabilities.

5.8 Putting It All Together

Having the new probabilities in place we can iterate the above steps:

```
;; Strings together prediction, summing, softmax-ing, and computing
;; the gradient
(define (create-fields dataset iteration mod-ids)
  (let (nclasses (count mod-ids)
        pred-ds (batch-predict dataset iteration mod-ids)
        sum-ds (create-sums pred-ds nclasses iteration)
        prob-ds (create-softmax-probs sum-ds nclasses iteration))
    (compute-gradient prob-ds nclasses iteration)))
```

Here we assume that we start with trained models, and execute an entire iteration using the functions defined above, so that we come all the way back to the point where we can train models again.

All that's left is to iterate that function until a stopping condition, along with a few set up and tear down steps.

```
(define (gradient-boost dataset)
  (let (objective (dataset-get-objective-id dataset)
        inputs (default-inputs dataset objective)
        nclasses (get-num-classes dataset objective)
        formatted (format dataset nclasses inputs objective))
    (loop (ds formatted
           iteration 1
           total-imp 0
           imp-1 0
           imp-2 0
           models [])
      (log-info "Iteration " iteration)
      (let (sets (bootstrap ds iteration)
            train (sets 0)
            test (sets 1)
            last-gradient (sum-gradient test nclasses (- iteration 1))
            _ (log-info "Gradient: " last-gradient)
            new-models (learn-trees train nclasses (- iteration 1))
            new-test (create-fields test iteration new-models)
            this-gradient (sum-gradient new-test nclasses iteration)
            _ (log-info "Gradient: " this-gradient)
            this-imp (- last-gradient this-gradient)
            pct (* (/ (+ this-imp imp-1 imp-2) (+ this-imp total-imp)) 100))
        (log-info "Improvement over last 3 iterations: " pct "%")
        ;; Stop arbitrarily at 1% improvement over last three iterations
        (if (> pct 1)
          (recur (create-fields ds iteration new-models)
                 (+ iteration 1)
                 (+ total-imp this-imp)
                 this-imp
                 imp-1
                 (append models new-models))
          models)))))
```

It seems like a lot, but the main bits are the stopping condition (we keep track of the last three total gradient magnitudes and stop if they represent less than 1% of the total improvement) and also the fact that we learn over a bootstrap sample of our original dataset. That is, at each iteration we learn over some of the dataset and check improvement over the other part. As such, we have to call `create-fields` on both the training and the test sets

5.9 Conclusion

We've implemented a "vanilla" version of gradient tree boosting in WhizzML. Hopefully, we've proven along the way that it's possible to implement many complex machine learning algorithms in WhizzML, and thereby gain the power of BigML's infrastructure behind your implementation.

CHAPTER 6

Anomaly-Based Covariate Shift

Detecting Covariate Shift and Dataset Shift in new production data relative to previously trained predictive models is always a challenge. BigML has discussed a method for doing this[1] based on the Matthews Correlation Coefficient (Phi Coefficient) computed from the confusion matrix for a predictive model. A tutorial also describes a WhizzML package for implementing this method (see the "Covariate Shift" section). This tutorial describes an alternative method that can be easily implemented using WhizzML that can be useful for some types of data.

In the alternative method we describe here, we use the WhizzML anomaly detection functions. The method computes an average anomaly score of the production dataset relative to the model training dataset as a measure of the covariate shift between the training dataset and the production dataset. An anomaly detector is trained from the same dataset used to train the model. This anomaly detector is then used to derive a batch anomaly score for the production dataset. Finally, the average value of that batch anomaly score is computed as an indicator of covariate shift.

The full code for this tutorial is available in our whizzml examples repository[2] on Github.

6.1 Code Overview

We first review the functions in the package to explain the anomaly-based approach to covariate-shift estimation.

```
(define (sample-dataset dst-id rate oob seed)
  (create-and-wait-dataset {"sample_rate" rate
                            "origin_dataset" dst-id
                            "out_of_bag" oob
                            "seed" seed}))
```

Inputs:

- `dst-id`: (`string`) ID of the dataset to be sampled.
- `rate`: (`float`) A value between 0 and 1 that specifies the size of the bagged sample. For example, 0.8 means that 80% of original dataset is in the bagged sample.
- `oob`: (`boolean`) Selects whether we want the bagged (false) chunk of data or the out of bag (true) chunk. For example, if the `rate` is 0.75, and `oob` is false, we get 75% of the data. If `oob` is true, we get the other 25%.
- `seed`: (`string`) A string used to make the sampling deterministic (repeatable)

Output: (`string`) ID of the new dataset object.

[1] https://blog.bigml.com/2014/01/03/simple-machine-learning-to-detect-covariate-shift/
[2] https://github.com/whizzml/examples/tree/master/anomaly-shift

To build a practical anomaly-based covariate shift detector, we need to derive sample from the anomaly detector training dataset and the production dataset. This helper routine wraps the basic dataset creation function to isolate configuration of that function so that it can be easily modified to support other sampling options in other applications. This basic version simply creates a deterministic sample determined by `seed` parameter of a size determined jointly by `rate` and `oob` from the source dataset specified by `dst-id`.

```
(define (anomaly-evaluation anomaly-id dst-id)
  (create-and-wait-batchanomalyscore {"anomaly" anomaly-id
                                      "dataset" dst-id
                                      "all_fields" true
                                      "output_dataset" true }))
```

Inputs:

- `anomaly-id`: (`string`) ID of the anomaly detector object.
- `dst-id`: (`string`) ID of the production dataset object.

Output: (`string`) ID of the created batchanomalyscore object.

We also will need to apply an anomaly detector derived from a training dataset to a production dataset. As with the `sample-dataset`, this helper routine wraps the WhizzML batch anomaly score computation function to isolate the configuration information so that it can be easily modified to use other evaluation options. This basic version simply applies the anomaly detector specified by `anomaly-id` to the dataset specified by `dst-id` and returns a BigML batchanomalyscore object. Because we specify `all_fields` and `output_dataset` as `true` in the WhizzML function to create the batchanomalyscore object, the returned metadata includes a reference to a dataset that includes the original production dataset with each item annotated with an additional member that contains the anomaly score for the item and additional metadata that includes the anomaly score results for the whole dataset.

```
(define (avg-anomaly evdst-id)
  (let (evdst (fetch evdst-id)
        score-field (evdst ["objective_field" "id"])
        sum (evdst ["fields" score-field "summary" "sum"])
        population (evdst ["fields" score-field "summary" "population"]))
    (/ sum population)))
```

Inputs:

- `evdst-id`: (`string`) ID of the batch anomaly score dataset object.

Output: (`float`) Average batch anomaly score from the results in the batch anomaly score dataset object.

Finally, we encapsulate the computation of the average anomaly score for the production dataset as a helper function that computes the average anomaly score from the annotated production dataset object returned by `anomaly-evaluation`. The metadata for this augmented production dataset object includes a `["objective_field" "id"]` member that contains the name of the `score-field` for the member with the object that has summary anomaly score results for the production dataset. The quotient of the members `["fields" score-field "summary" "sum"]` and `["fields" score-field "summary" "population"]` in that anomaly score member is returned as the average batch anomaly score for the production dataset object `evdst-id`.

```
(define (anomaly-measure train-dst train-exc prod-dst prod-exc seed clean)
  (let (traino-dst (sample-dataset train-dst 0.8 false seed)
        prodo-dst (sample-dataset prod-dst 0.8 true seed)
        anomaly (create-and-wait-anomaly {"dataset" traino-dst
                                          "excluded_fields" train-exc})
        ev-id (anomaly-evaluation anomaly prodo-dst)
        evdst-id ((fetch ev-id) ["output_dataset_resource"])
        _ (wait evdst-id)
```

```
        score (avg-anomaly evdst-id))
      (if clean
        (prog (delete evdst-id)
              (delete ev-id)
              (delete anomaly)
              (delete prodo-dst)
              (delete traino-dst)))
      score))
```

Inputs:

- `train-dst`: (`string`) ID of the training dataset.
- `train-exc`: (`list`) Fields to exclude from the training dataset.
- `prod-dst`: (`string`) ID of the production dataset.
- `prod-exc`: (`list`) Fields to exclude from the production dataset.
- `seed`: (`string`) A string used to make the sampling deterministic (see the "sample-dataset" function).
- `clean`: (`boolean`) Delete intermediate datasets before exiting the function.

Output: (`float`) The average anomaly score between 0 and 1.

This top-level function combines the previous helper functions to compute the average anomaly score for a single production dataset relative to an anomaly detector built from the training dataset.

The function accepts a training dataset `train-dst` and a production dataset `prod-dst` along with a `seed` string to force deterministic sampling of both. 80% of the training dataset `traino-dst` is used to train an anomaly detector `anomaly`, ignoring the fields in the list `train-exc`. 20% of the production dataset `prodo-dst` is then evaluated with the anomaly detector to produce a BigML batchanomalyscore object `ev-id`. The batchanomalyscore `ev-id` metadata has the ID `evdst-id` for the annotated version of the production dataset. The metadata for this dataset includes the anomaly score information used by `avg-anomaly` to compute an average anomaly `score` for the sample of the production dataset.

Finally, if `clean` is specified as `true` the intermediate objects created by the function are deleted before the function returns the `score`.

```
(define (anomaly-loop train-dst
                      train-exc
                      prod-dst
                      prod-exc
                      seed
                      niter
                      clean
                      logf)
  (loop (iter 1
         scores-list [])
    (if logf
      (log-info "Iteration " iter))
    (let (score (anomaly-measure train-dst
                                 train-exc
                                 prod-dst
                                 prod-exc
                                 (str seed " " iter)
                                 clean)
          scores-list (append scores-list score))
      (if logf
        (log-info "Iteration " iter scores-list))
      (if (< iter niter)
        (recur (+ iter 1)
```

```
                scores-list)
        scores-list))))
```

Inputs:

- `train-dst`: (`string`) ID of the training dataset.
- `train-exc`: (`list`) Fields to exclude from the training dataset.
- `prod-dst`: (`string`) ID of the production dataset.
- `prod-exc`: (`list`) Fields to exclude from the production dataset.
- `seed`: (`string`) A string used to make the sampling deterministic (see the "sample-dataset" function).
- `niter`: (`number`) Number of iterations.
- `clean`: (`boolean`) Delete intermediate datasets before exiting the function.
- `logf`: (`boolean`) Enables logging.

Output: (`list`) A list of average anomaly scores between 0 and 1 for specified datasets over `niter` trials.

To facilitate logging and to illustrate the implementation of multiple train-evaluate iterations in way that can be easily augmented in specific applications, we implement iteration over the `anomaly-measure` as an explicit `loop` function.

The loop function inputs are an iteration count `iter` and a list of scores computed thus far `scores-list`. Each iteration of the loop generates a unique `seed` value for sampling the training and production datasets and computes the anomaly `score` with `anomaly-measure`. That `score` is appended to the `scores-list`. If `niter` iterations have not been completed, the iteration count is updated and the loop repeated.

```
(define (anomaly-measures train-dst
                          train-exc
                          prod-dst
                          prod-exc
                          seed
                          niter
                          clean
                          logf)
  (let (values (anomaly-loop train-dst
                             train-exc
                             prod-dst
                             prod-exc
                             seed
                             niter
                             clean
                             logf))
    values))
```

Inputs:

- `train-dst`: (`string`) ID of the training dataset.
- `train-exc`: (`list`) Fields to exclude from the training dataset.
- `prod-dst`: (`string`) ID of the production dataset.
- `prod-exc`: (`list`) Fields to exclude from the production dataset.
- `seed`: (`string`) A string used to make the sampling deterministic (see the "sample-dataset" function).
- `niter`: (`number`) Number of iterations.
- `clean`: (`boolean`) Delete intermediate datasets before exiting the function.

- `logf`: (`boolean`) Enables logging.

Output: (`list`) A list of average anomaly scores between 0 and 1 for the specified datasets over `niter` trials.

This top-level function computes a list of `niter` average anomaly scores for pairs of samples of the training dataset `train-dst` and the production dataset `prod-dst`. For illustration purposes we implement this function as a wrapper function around the `anomaly-loop` function that does the actual work and pass all of the input values including those mentioned and `train-exc`, `prod-exc`, `seed`, `clean`, and `logf` directly to that function. For other applications, alternative looping structures could be implemented.

```
(define (anomaly-estimate train-dst
                          train-exc
                          prod-dst
                          prod-exc
                          seed
                          niter
                          clean
                          logf)
  (let (values (anomaly-measures train-dst
                                  train-exc
                                  prod-dst
                                  prod-exc
                                  seed
                                  niter
                                  clean
                                  logf)
        sum (reduce + 0 values)
        cnt (count values))
    (/ sum cnt)))
```

Inputs:

- `train-dst`: (`string`) ID of the training dataset.
- `train-exc`: (`list`) Fields to exclude from the training dataset.
- `prod-dst`: (`string`) ID of the production dataset.
- `prod-exc`: (`list`) Fields to exclude from the production dataset.
- `seed`: (`string`) A string used to make the sampling deterministic (see the "sample-dataset" function).
- `niter`: (`number`) Number of iterations.
- `clean`: (`boolean`) Delete intermediate datasets before exiting the function.
- `logf`: (`boolean`) Enables logging.

Output: (`float`) The average anomaly scores between 0 and 1 for the specified datasets for the `niter` trials.

Finally, in this top-level function we use `anomaly-measures` to compute a list of `niter` average anomaly scores for pairs of samples of the training dataset `train-dst` and the production dataset `prod-dst`. We then compute the average of the scores in that list and return that as a single measure of the covariate shift between the training dataset `train-dst` and the production dataset `prod-dst`.

6.2 Examples

We end this tutorial description with a few examples of how to use the top-level functions.

If we only want an estimate of the data shift computed from a single pair of samples from the training dataset and the production dataset, we can use the top-level function:

```
(define cvshift-one (anomaly-measure "dataset/..."
                                     []
                                     "dataset/..."
                                     []
                                     "test-run-1"
                                     false))
```

In this example we don't exclude any of the fields from the training dataset and specify that we would like to preserve all of the intermediate objects created in the computation by specifying `clean` as `false`.

Suppose next that we are concerned our training or production datasets aren't uniform in some sense. Suppose also that the datasets include a dependent variable `idvar` that differs consistently between the two datasets potentially affecting the anomaly score. In this case, we can use the top-level function that generates a series of anomaly scores:

```
(define cvshift-set (anomaly-measures "dataset/..."
                                      ["idvar"]
                                      "dataset/..."
                                      ["idvar"]
                                      "test-run"
                                      20
                                      true
                                      true))
```

Here we specify we want a series of 20 train-evaluate iterations and that logging information should be generated by specifying `logf` as `true`. To manage storage on our BigML account we also define `clean` as `true` to delete all intermediate objects each iteration.

Finally, we can run a series of train-evaluate trials and then compute the average of the anomaly scores using the final top-level function as:

```
(define cvshift-avg (anomaly-estimate "dataset/..."
                                      ["idvar"]
                                      "dataset/..."
                                      ["idvar"]
                                      "test-run"
                                      20
                                      true
                                      true))
```

In this case we specify the same sequence of anomaly score computations and then take the average returned by the function as our estimate of covariate-shift between the training dataset `train-dst` and the production dataset `prod-dst`.


bigml